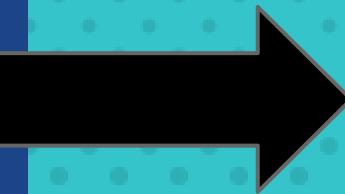


POLIMORFISMO

PARTE 2

Programação Orientada a Objetos



Abstração

Encapsulamento	Construtores	Objetos	Métodos	Atributos	Classes	Herança
	Modificadores de acesso	Métodos de acesso	Métodos modificadores	Getters e Setters	Métodos estáticos	Herança
Polimorfismo	Reutilização	Sobrescrita de métodos	Associação	Extends	Super e subclasse	Herança
	Classes abstratas	instanceof e as operator	métodos abstratos	interfaces	Generics	END

CLASSES ABSTRATAS

POLIMORFISMO

A definição vem de poli (muitas) morfismo (formas), e trata-se de termos **comportamentos diferentes para um único método**

Vamos analisar o projeto do game para identificar comportamentos polimórficos

Ex.:

```
const x: Personagem = new Guerreiro('Gladimir');
```

```
const y: Personagem = new Priest('Edécio');
```

```
x.atacar(y);
```

```
y.atacar(x);
```

INSTANCEOF

A palavra-chave `instanceof` pode ser utilizada para identificar se uma classe é ou não de uma especialização.

```
const x: Personagem = new Guerreiro('Gladimir');
if(x instanceof Guerreiro){
    // Executar ações específicas de guerreiro
    x.umMetodoDoGuerreiro()
}
```

E AS CLASSES ABSTRATAS?

CLASSE ABSTRATAS

Muitas vezes é útil se ter classes em que **não se pode instanciar**

```
const x: Personagem = new Personagem();
```

Instanciar um Personagem não faz sentido, já que precisamos das especializações para executar as tarefas

Uma classe abstrata é uma classe **incompleta**, que não pode ser instanciada e **pode possuir métodos abstratos**

CLASSE ABSTRATAS

Para declararmos uma classe abstrata basta adicionar um **abstract** antes da palavra-chave `class` na declaração da mesma.

```
abstract class Personagem{...}
```

Outro **problema**, ou no mínimo uma característica estranha da nossa aplicação, é que tivemos que desenvolver métodos na superclasse que estão ali apenas para que possamos **sobrescrevê-los** nas subclasses.

CLASSE ABSTRATAS

Se esses métodos não existissem, não poderíamos invocar estes métodos através da superclasse Personagem.

```
const p: Personagem = new Guerreiro('Dedé');
p.atacar(oponente);
```

Como o objeto foi tipado com Personagem, apenas verá o que está em Personagem.



```
1 export class Personagem {
2   constructor(
3     protected _nome: string,
4     protected _forca: number,
5     protected _habilidadeMental: number,
6     protected _podeDeAtaque: number,
7     protected _esquiva: number,
8     protected _resistencia: number,
9     protected _vidaAtual: number,
10    protected _vidaMaxima: number
11  ) {}
12
13  public atacar(personagem: Personagem): void {
14    console.log("Um comportamento desconhecido");
15  }
16
17  public contraAtacar(personagem: Personagem): void {
18    console.log("Comportamento desconhecido");
19  }
20
21  public aprimorarHabilidadePrincipal(): void {
22    console.log("Comportamento desconhecido");
23  }
24
25  public regenerarVida(): void {
26    console.log("Comportamento desconhecido");
27  }
28
29 }
```

MÉTODOS ABSTRATOS E INTERFACES

MÉTODOS ABSTRATOS

Outra característica das classes abstratas é que elas podem possuir métodos abstratos, que **são métodos que possuem apenas sua assinatura declarada**, assim não precisamos fazer gambiarras para sobrescrevê-los.

Poderíamos aplicar essa funcionalidade nos nossos métodos de atacar, contra-atacar, aprimorar habilidade e regenerar vida!

Um método abstrato **obrigatoriamente** deve ser implementado para subclasse concreta da superclasse abstrata

MÉTODOS ABSTRATOS - EXEMPLO

```
public abstract atacar(Personagem personagem): void;
```

Em vez de:

```
public atacar(Personagem personagem): void{  
    console.log("Um comportamento desconhecido");  
}
```

```
1 export class Personagem {  
2     constructor(  
3         protected _nome: string,  
4         protected _forca: number,  
5         protected _habilidadeMental: number,  
6         protected _poderDeAtaque: number,  
7         protected _esquiva: number,  
8         protected _resistencia: number,  
9         protected _vidaAtual: number,  
10        protected _vidaMaxima: number  
11    ) {}  
12  
13    public atacar(personagem: Personagem): void {  
14        console.log("Um comportamento desconhecido");  
15    }  
16  
17    public contraAtacar(personagem: Personagem): void {  
18        console.log("Comportamento desconhecido");  
19    }  
20  
21    public aprimorarHabilidadePrincipal(): void {  
22        console.log("Comportamento desconhecido");  
23    }  
24  
25    public regenerarVida(): void {  
26        console.log("Comportamento desconhecido");  
27    }  
28}  
29
```



```
1 export abstract class Personagem {  
2     constructor(  
3         protected _nome: string,  
4         protected _forca: number,  
5         protected _habilidadeMental: number,  
6         protected _poderDeAtaque: number,  
7         protected _esquiva: number,  
8         protected _resistencia: number,  
9         protected _vidaAtual: number,  
10        protected _vidaMaxima: number  
11    ) {}  
12  
13    public abstract atacar(personagem: Personagem): void;  
14  
15    public abstract contraAtacar(personagem: Personagem): void;  
16  
17    public abstract aprimorarHabilidadePrincipal(): void;  
18  
19    public abstract regenerarVida(): void;
```

AO TENTAR INSTANCIAR

```
}
```

```
const person: Personagem = new Personagem();
```

Não é possível criar uma instância de uma classe abstrata. ts(2511)

[Exibir o Problema \(F8\)](#) Nenhuma correção rápida disponível

POLIMORFISMO

INTERFACES

INTERFACE

Utilizando interfaces é o mecanismo utilizado pelo typescript para proporcionar um comportamento similar ao da herança múltipla



Se liga!

Assim como na herança, podemos declarar uma variável do **tipo da interface**, e atribuir a ela uma classe concreta.

INTERFACE - CARACTERÍSTICAS

Os métodos **devem ser** públicos e abstratos

Atributos apenas se forem **constantes**

Um classe pode implementar várias interfaces

Quando se herda uma classe se usa extends, quando se implementa uma interface, se usa **implements**

**E SE UMA CLASSE QUE
IMPLEMENTA UMA INTERFACE NÃO
IMPLEMENTAR TODOS OS
MÉTODOS ABSTRATOS DESTA?**

INTERFACE - SINTÁXE

```
public interface Calculadora{  
    somar(v1: number, int: number): number;  
    subtrair(v1: number, v2: number): number;  
}
```

Note!

Automaticamente esses métodos já são **public abstract**

O class é substituído por **interface**

Basicamente, uma classe abstrata que não pode possuir métodos concretos.

INTERFACE - APLICAÇÕES

Especificar o que fazer mas sem dizer COMO fazer.

Ex.:

Ações dos nossos personagens

Operações de Banco de dados

Objetos em Games

Tipos de calculadora

NO NOSSO PROJETO - INTERFACE DAO



```
1 export interface PersonagemDAO {  
2     salvar(personagem: Personagem): Personagem;  
3  
4     remover(idPersonagem: number): Personagem;  
5  
6     listar(): Personagem[];  
7  
8     atualizar(personagem: Personagem): Personagem;  
9 }
```

NO NOSO PROJETO - PERSONAGEM

```
export abstract class Personagem implements PersonagemDAO{  
    constructor(  
        protected _nome: string,  
        protected _forca: number,  
        protected _habilidades: string[],  
        protected _poderDeAtaque: number,  
        protected _esquiva: number,  
        protected _resiste: number,  
        protected _vidaAtual: number,  
        protected _vidaMaxima: number  
    ) {}  
  
    public abstract atacar(personagem: Personagem): void;  
}
```

Personagem implements PersonagemDAO{

class Personagem

A classe 'Personagem' implementa incorretamente a interface 'PersonagemDAO'.

O tipo 'Personagem' não tem as propriedades a seguir do tipo 'PersonagemDAO': salvar, remover, listar, atualizar ts(2420)

[Exibir o Problema \(F8\)](#) [Correção Rápida... \(⌘.\)](#)

NO NOSSO PROJETO - PERSONAGEM

O problema somente se resolve depois que a classe implementa **TODOS** os métodos da interface.

É uma maneira de **forçar** padrões e implementações necessários para alguns dados.



```
1 export abstract class Personagem implements PersonagemDAO{  
2     constructor(  
3         protected _nome: string,  
4         protected _forca: number,  
5         protected _habilidadeMental: number,  
6         protected _poderDeAtaque: number,  
7         protected _esquiva: number,  
8         protected _resistencia: number,  
9         protected _vidaAtual: number,  
10        protected _vidaMaxima: number  
11    ) {}  
12    salvar(personagem: Personagem): Personagem {  
13        throw new Error("Method not implemented.");  
14    }  
15    remover(idPersonagem: number): Personagem {  
16        throw new Error("Method not implemented.");  
17    }  
18    listar(): Personagem[] {  
19        throw new Error("Method not implemented.");  
20    }  
21    atualizar(personagem: Personagem): Personagem {  
22        throw new Error("Method not implemented.");  
23    }
```

NO NOSSO PROJETO - PERSONAGEM

Um interface pode estender outra interface.

```
● ● ●  
1 interface PersonagemDAOAdvanced extends PersonagemDAO {  
2     buscarPorID(idPersonagem: number): Personagem;  
3  
4     buscarPorNome(nome: string): Personagem;  
5 }
```

NO NOSSO PROJETO - PERSONAGEM

Uma classe pode implementar N interfaces.

```
● ● ●  
1 export abstract class Personagem implements PersonagemDAO, OutraInterface {  
2   constructor(  
3     protected _nome: string,  
4     protected _forca: number,  
5     protected _habilidadeMental: number,  
6     protected _poderDeAtaque: number,  
7     protected _esquiva: number,  
8     protected _resistencia: number,  
9     protected _vidaAtual: number,  
10    protected _vidaMaxima: number  
11  ) {}
```

NO NOSSO PROJETO - PERSONAGEM

A interface também serve para tipar objetos.

```
const personagemDAO: PersonagemDAO = new Guerreiro("Uhtred")
personagemDAO.|
    □ atualizar
    □ listar
    □ remover
    □ salvar
```

CAST

Personagem instanciado e
vendo todos os seus
métodos

```
const personagem: Personagem = new Guerreiro("Uhtred");
personagem.
(personagem ⚡ aprimorarHabilidadePrincipal
    ⚡ atacar
    ⚡ atualizar
    ⚡ contraAtacar
    ⚡ esquiva
    ⚡ listar
    ⚡ nome
    ⚡ poderDeAtaque
    ⚡ regenerarVida
    ⚡ remover
    ⚡ resistencia
    ⚡ resumo
```

Se eu quiser "converter" ele para pegar
apenas os métodos da interface

```
const personagem: Personagem = new Guerreiro("Uhtred");
(personagem as PersonagemDAO).atualizar(personagem: Personagem): Personagem
```

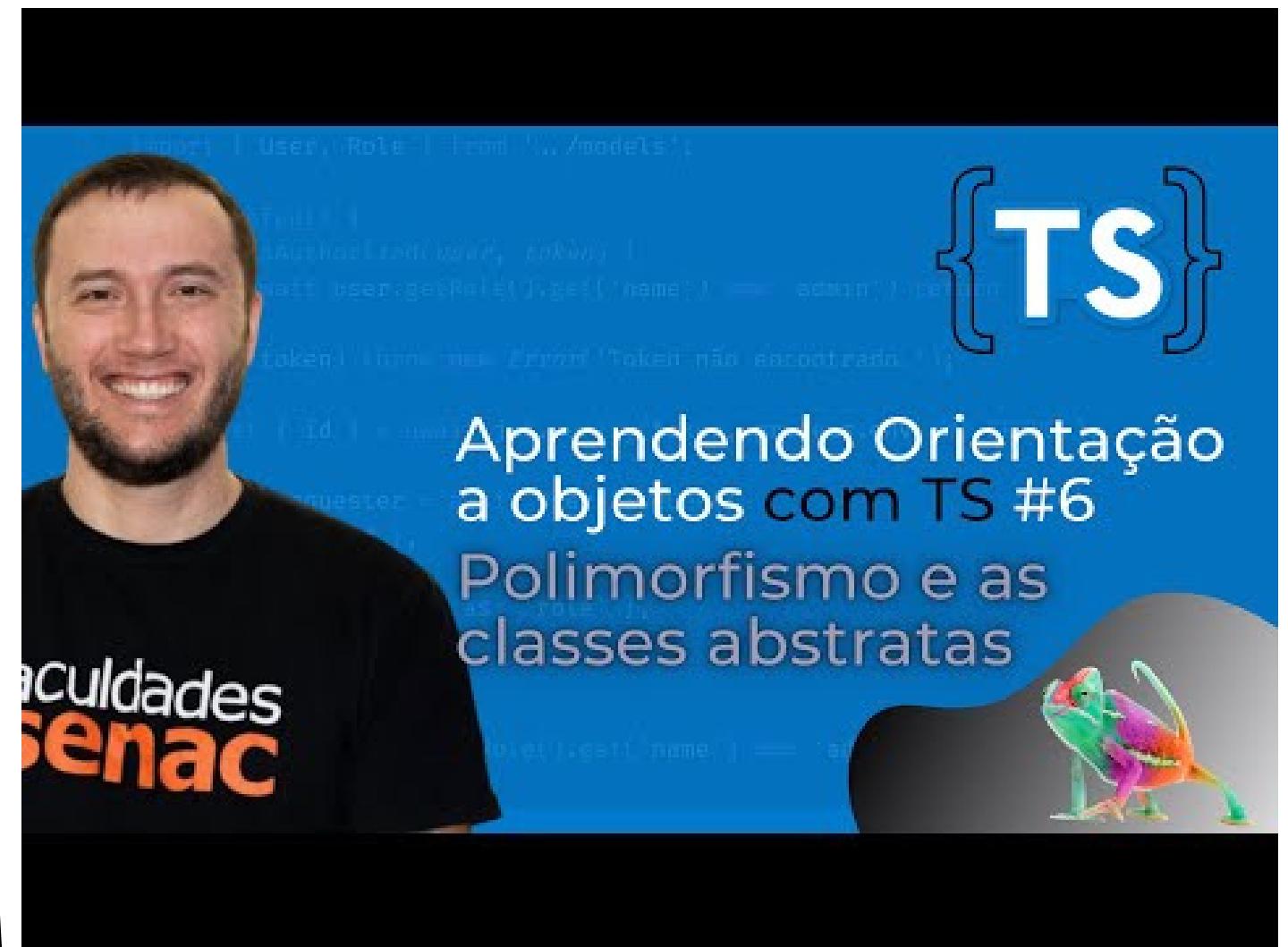
CÓDIGO-FONTE

Repositório github



MATERIAL DE APOIO

Conteúdo sobre classes abstratas



Proposta de trabalho:

[**AQUI**](#)